Hard Multidimensional Multiple Choice Knapsack Problems, an Empirical Study

Bing Han[*,a,b], Jimmy Leblet[a], Gwendal Simon[a]

[a] *Department of Computer Science, Institut Telecom - Telecom Bretagne, Technople Brest-Iroise, 29238 Brest, France*
[b] *Department of Computers and Networks, Institut Telecom - Telecom ParisTech, 37/39, rue Dareau, 75014 Paris, France*

Abstract

Recent advances in algorithms for the multidimensional multiple choice knapsack problems have enabled us to solve rather large problem instances. However, these algorithms are evaluated with very limited benchmark instances. In this study, we propose new methods to systematically generate comprehensive benchmark instances. Some instances with special correlation properties between parameters are found to be several orders of magnitude harder than those currently used for benchmarking the algorithms. Experiments on an existing exact algorithm and two generic solvers show that instances whose weights are uncorrelated with the profits are easier compared with weakly or strongly correlated cases. Instances with classes containing similar set of profits for items and with weights strongly correlated to the profits are the hardest among all instance groups investigated. These hard instances deserve further study and understanding their properties may shed light to better algorithms.

Key words: multidimensional, multiple choice, knapsack problem, algorithm, performance evaluation

[*] Corresponding author.
Email addresses: bing.han@telecom-bretagne.eu (Bing Han), jimmy.leblet@telecom-bretagne.eu (Jimmy Leblet), gwendal.simon@telecom-bretagne.eu (Gwendal Simon)

1. Introduction

Multidimensional Multiple choice Knapsack Problem (MMKP) is one of the most complex members of the Knapsack Problem (KP) family. It could be stated as follows: We are given m classes with each class i containing n_i items. The jth item of class i has profit p_{ij}. Each item has l dimensions of weight, and the weight at dimension k is denoted as w_{ijk}. The knapsack has capacity c_k on each dimension k. The goal is to select one item in each class to maximize the sum of their profits and to keep the total weight on each dimension no more than the corresponding capacity. It is generally considered that the profits, weights and the knapsack capacities take non-negative values, thus we will not explicitly state this constraint in the formulation. Formally, MMKP could be expressed with an integer programming model:

$$\text{(MMKP)} \quad \text{maximize} \quad \sum_{i=1}^{m}\sum_{j=1}^{n_i} p_{ij}x_{ij} \tag{1}$$

$$\text{subject to} \quad \sum_{i=1}^{m}\sum_{j=1}^{n_i} w_{ijk}x_{ij} \leq c_k, k = 1, \ldots, l \tag{2}$$

$$\sum_{j=1}^{n_i} x_{ij} = 1, i = 1, \ldots, m \tag{3}$$

$$x_{ij} \in \{0, 1\}, i = 1, \ldots, m, j = 1, \ldots, n_i \tag{4}$$

where the binary variable x_{ij} indicates the jth item of class i is selected or not. For clarity, we assume all classes have the same number of items in this study, *i.e.* $n_1 = \cdots = n_m = n$.

MMKP has many applications. It has been used to model the Quality of Service (QoS) management problem in computer networks [1] and the admission control problem in the adaptive multimedia systems [2, 3, 4]. Various other resource allocation problems can also be mapped directly to MMKP, please refer to [5, 6] and the references wherein.

Most academic efforts related with MMKP have been put on finding heuristic algorithms due to the NP-hard nature of the problem [4, 7, 8, 9]. However,

computing exact MMKP solutions can also be of interest when the computation time constraint is not critical, *e.g.* exact solution is the most valuable benchmark for the heuristic algorithms. More important, when the exact algorithms can solve most problem instances quickly and only fall into exponential time in rare cases, they can be applied to some practical problems. Thus, for any exact algorithm, it is important to know how often it falls into the trap.

Great efforts have been taken in analyzing the structure of many KP family members, *e.g.* simple KP, Bounded KP (BKP), Multiple KP (MKP), Multiple Choice KP (MCKP), Multidimensional KP (MDKP) *etc.* Comprehensive discussions on these problems could be found in [10, 5]. It has been demonstrated that all of them are highly structured. Exploiting their special structural properties usually leads to efficient algorithms that are able to solve certain category of problem instances in reasonable time, although the problems are NP-hard. Moreover, it appears that some instances are particularly hard to solve not because of the size of the problem (number of input variables or the magnitude of the variable values), but because of the special combination of variable values, *i.e.* even a small problem instance can require a long time to solve. Generally, the relationship between the profits and weights of the items plays a very important role in the solution time of certain problem instances. Besides, a very important observation has been made on simple KP that the relationship between the capacity and the weight also has great impact on the hardness of the instances [11].

In contrast, little work has been done on MMKP in analyzing its structure. To the best of our knowledge, besides many proposed algorithms, the only theoretical analysis is presented in [12]. The authors show that the proportion of the dominated variables can be estimated as a probability function of the number of dimensions and the number of items in each class. The results could be used to reduce the problem size during the pre-process stage. However, it is still unclear how the parameters such as profit, weight and capacity interact in making the problem difficult. Furthermore, almost all MMKP algorithms proposed in the literature are evaluated against very limited problem instances

which may belong to special easy cases.

In this paper, we study the relationship between various parameters of the MMKP such as profit, weight and capacity in order to identify the key factors that make a hard instance. Furthermore, uncorrelated, weakly correlated and strongly correlated cases for items within each class, between classes and across multiple dimensions are investigated. To the best of our knowledge, no such work has been reported in the current literature. A systematic method to generate comprehensive MMKP test instances is proposed. Several groups of instances generated with the proposed methods are evaluated with the BBLP algorithm [3, 4] and two integer programming solvers, ILOG CPLEX [13] and GLPK [14]. The experiments show that many instances are several orders of magnitude harder than traditionally used ones in terms of the computing time. These hard MMKP instances usually have medium knapsack capacity and high correlation between weights and profits. The experiments also suggest that instances with similar set of profits across classes and with strong correlations between weights and profits are hard to solve.

In the rest of this paper, we first give a brief survey of existing MMKP algorithms and benchmarking methods in Section 2 and Section 3, respectively. Then in Section 4, we propose new methods to generate MMKP benchmark instances. Section 5 is dedicated to evaluating the difficulty of these instances using the existing exact algorithm and solvers. Finally, we draw conclusion and propose important future works in Section 6.

2. Existing MMKP Algorithms

Several heuristic algorithms have been proposed for MMKP. The first heuristic is proposed by Moser *et al.* [7] based on Lagrange multiplier method. Then HEU is proposed by Khan *et al.* [3, 4]. Later, Hifi *et al.* proposed MGLS in [15] and MRLS in [8]. A convex hull based method was proposed by Akbar *et al.* [9]. Two algorithms based on the column generation method have been proposed recently by Cherfi and Hifi [16]. These heuristics are able to obtain fairly

good solutions for large MMKP instances. In comparison, exact algorithms have not received a lot of attention. Indeed, only two exact algorithms have been proposed in the literature, both are based on the branch-and-bound principle.

A straightforward extension of the branch-and-bound method for KP to MMKP is the BBLP algorithm [3, 4]. BBLP starts from a state that all variables are undecided. At each round, BBLP selects a class i and generates branches with each item in the class. This step corresponds to assigning value 1 to the binary variable of the selected item and assigning value 0 for the others. As a result, n_i partial solutions will be generated. Infeasible partial solutions are dropped immediately while feasible ones, including those obtained from all previous rounds, are examined and the one with the maximum upper bound is selected for branching in the next round. The upper bound is obtained by solving the LP relaxation of a sub-problem containing only the undecided variables. If at a certain round, the best solution selected has all variables fixed, an optimal solution is obtained.

Another exact algorithm is the EMKP algorithm [17]. EMKP initially sorts the items in each class in descending order of their profits. A tie between items is resolved by comparing the relative aggregated resource consumption of the items which is obtained by summing up the weight-capacity ratios across all dimensions. Then it starts the branch-and-bound procedure from the first item in the first class. Based on the current node, two nodes are further developed if they exist: a son node corresponding to selecting the first item of the next class and a brother node corresponding to selecting the next item in the same class. The bounding procedure employs an upper bound obtained with an auxiliary MCKP problem and a supplementary KP. The auxiliary MCKP is formed with weight and capacity in the original problem aggregated across multiple dimensions, and the supplementary KP is formed with all items not selected and the aggregated residual capacity. In order to further trim the branches of the search tree, a feasible solution obtained by the MRLS heuristic [8] is used as a lower bound. A node with a solution upper bound below the lower bound is dropped.

However, we have identified some fundamental problems of the EMKP al-

gorithm [17]. First of all, the EMKP algorithm employs basically a sequential search strategy, *i.e.* the development of a certain node depends on its previous brother nodes. This prevents pruning the infeasible nodes effectively as some feasible nodes may have to be developed from them. Secondly, the EMKP algorithm selects the *best* node to develop at each round, but the paper does not state how the selection is made. A common way is to select the node with the highest upper bound. However, for the infeasible nodes which are kept in the search tree, there is no natural way to calculate an upper bound. Finally, the sequential search strategy implicitly requires generating the brother node for every node who has a brother. This is true even for the unpromising nodes whose upper bound is below the lower bound. However, EMKP tries to reduce the search space by pruning these unpromising nodes (line 17 of the algorithm in [17]). This will break the searching process and the algorithm will not be able to find the optimal solution which have to be reached through the pruned node. Even worse, the better the quality of the lower bound is (approaching the optimal from below), the more often the improper prune may happen.

As a result, we will consider only the BBLP algorithm and two generic MIP solvers GLPK and CPLEX in this study.

3. Existing Methods to Generate Benchmark Instances

It is very important to test the algorithms with problem instances in order to know their performance in practice. When algorithms are tackling a particular problem, ideal instances for performance evaluation are those from traces of the real world. However, as MMKPs usually originate from a diverse applicative background, typical instances from a certain domain may hardly be reasonable for another. Moreover, there is no systematic report on real world MMKP problem instances in the literature. In contrast, test instances can be generated to cover a much wider range of instance types. As a result, generated instances play an important role in benchmarking the algorithms and have been used in most KP and MMKP related researches. In the following, we first describe

how the KP instances are usually generated, then we give a brief review on the current method to generate the MMKP instances. The latter is basically a straightforward extension of the former.

3.1. Generating KP instances

In order to generate a KP instance with certain number of items, the idea is to first assign values to both profit and weight of each item, then to set the capacity of the knapsack. Several groups of instances have been identified for KP considering the correlation between the profits and weights [18].

- *uncorrelated instances.* For this category, the profit of an item is independent to its weight. A commonly used method is to select profits (p_j) and weights (w_j) randomly in a certain interval, *e.g.* $[1, R]$. These instances are generally easy to solve.

- *weakly correlated instances.* In weakly correlated instances, the profit of an item is related with its weight, *e.g.* to select w_j randomly in $[1, R]$ and p_j in $[w_j - R/10, w_j + R/10]$ while ensuring $p_j \geq 1$.

- *strongly correlated instances.* For these instances, the profit of an item is a linear function of its weight plus a certain shift, *e.g.* to select w_j randomly in $[1, R]$, but let $p_j = w_j + R/10$. This category of KP instances are generally hard to solve.

- *sub-set sum instances.* For this category of instances, the profit of an item is a linear function of its weight. As a result, only weight need to be considered when packing the knapsack.

Instances with other types of correlation can be defined similarly [18].

Finally, the capacity of the knapsack is set to a certain percentage of the sum weight. However, this has been shown to be inadequate as it generates the easiest KP instances under certain situations [11]. Thus the idea consists of generating a series of S test instances with the capacity c of the hth instance

selected as:

$$c = \frac{h}{S+1} \sum_{j=1}^{n} w_j. \tag{5}$$

3.2. Generating MMKP instances

To generate an MMKP instance with a given number of classes, a given number of items in each class and a given number of dimensions, the problem is to assign a profit value to each item, a weight value to each item at each dimension, and finally, a capacity value to each dimension of the knapsack. This can be done in various ways.

In [4], weights of items are uniformly selected in the interval $[0, R]$ indifferent to the dimensions or which class the item belongs to, where R denotes the maximum resource consumption by an item. Let P denote the maximum cost of unit resource, then the value $R \times P$ could be considered as the maximum cost (weight) of an item. Then uncorrelated instances are generated with profits assigned according to the item index in the class:

$$p_{ij} = \mathcal{U}\left(0, l \times \frac{R}{2} \times \frac{P}{2}\right) \times \frac{j+1}{n_i}. \tag{6}$$

For correlated instances, the profit is a linear function of the sum weight:

$$p_{ij} = \sum_{k=1}^{l} P_k w_{ijk} \times \mathcal{U}\left(0, l \times 3 \times \frac{R}{10} \times \frac{P}{10}\right), \tag{7}$$

where $P_k = \mathcal{U}(0, P)$ is a uniform random number between 0 and P.

Finally, the capacity of dimension k is set to half of the maximum possible resource consumption:

$$c_k = \frac{1}{2} \times m \times R. \tag{8}$$

The same set of instances have been used in [3, 6, 9, 19]. Instances generated with the same principle have been used in [15, 8, 16]. These instances are available at the *OR Benchmark Library* [20].

In [17], test instances are generated as follows: p_{ij} and w_{ij} are randomly selected in intervals $[0, 150]$ and $[0, 50]$, respectively. Capacities are set as:

$$c_k = \frac{1}{2} \sum_{i=1}^{m} \left(w_{ik}^{\min} + w_{ik}^{\max}\right), \tag{9}$$

where

$$w_{ik}^{\min} = \min_{1 \le j \le n} \{w_{ijk}\}, \tag{10}$$

$$w_{ik}^{\max} = \max_{1 \le j \le n} \{w_{ijk}\}. \tag{11}$$

In [21], domain related values are considered in the test instances. The number of classes, items in each class and dimensions are set according to a typical Video on Demand (VoD) system. The variable values are also set with respect to the typical values of a VoD system. For example, the weight of each item is set to the typical resource consumption of a session. The value is then scaled by a random number chosen from the interval $[0.75, 1.25]$ to mimic system dynamics. Similarly, the capacities are set to typical available resources scaled by a random value chosen from the interval $[0.95, 1.05]$.

Although these instances have been widely used in the literature, our computational results show that they are insufficient in demonstrating the performance of the algorithms. Table 1 presents the time used to solve the first few instances in the OR benchmark library with CPLEX, GLPK and the BBLP algorithm. Here we emphasize on the relative solution time across the instances. Notably, instances I3 and I4 take much more time than I5 and I6, despite they are smaller than the latter. This actually implies that not only the size of the instance, but also the structure play very important role in the solution time.

4. New Methods to Generate MMKP Problem Instances

Experiences from KP suggest that the correlation between profits and weights is critical to the hardness of an instance. Extending this idea to MMKP, we need to handle the correlation between the profits and multiple dimensions of weights. One direct way is to select the profit for each item then select the weights according to the profits. Given the number of classes (m), the number of items in each class (n, assuming all classes have the same number of items), and the number of dimensions (l), the MMKP is denoted as $P(m, n, l)$. We will also refer to a mapping from a class of n items to n values informally as a *generating function*.

4.1. Generating the Profits

In order to select the profits for items in each class i, we first bound the profits with two parameters $p_i^{\min}$ and $p_i^{\max}$ and select profit values within the interval. This could be done in various ways and here we define some generating functions for the profits.

Uniform Generating Function. Uniform random profits are natural in many real world problems and are widely used in the literature. In uniform generating function, we draw profit uniformly and randomly within the interval. We denote the uniform generating function as:

$$p_{ij} = \mathcal{U}\left(p_i^{\min}, p_i^{\max}\right). \tag{12}$$

Linear Generating Function. Items with linear profits are less studied in the literature. However, this kind of profit value assignment is actually quite common. For example in the QoS adaption problem [21], the QoS levels are usually mapped to the profit of items and their values are often consecutive integers. Also in the multi-hop query allocation problem [22], the query range is mapped to the profit and is measured in hop numbers which take also consecutive integer values.

In the linear generating function, we assign p_{ij} with a linear function of the item index j, *i.e.*

$$p_{ij} = \frac{j-1}{n_i - 1}\left(p_i^{\max} - p_i^{\min}\right) + p_i^{\min}. \tag{13}$$

For clarity, we use a short hand notation for this linear generating function as follows:

$$p_{ij} = \mathcal{L}\left(p_i^{\min}, p_i^{\max}\right). \tag{14}$$

Applying the Profit Generating Functions. The generating functions should be applied on each single class to generate the profits. Obviously, one can apply the same function to all classes or change the functions for each class. For the uniform generating function, even when it is applied to all classes with the same parameters, the random nature of the function will give different values

for profits in different classes. On the contrary, when the linear generating function is applied to all classes with the same parameters, all classes will have the same profit vector for their items. Therefore, besides applying the same generating function to all classes, we further propose two ways to use the generating functions. The first one is to reproduce the random profit vector generated by a uniform generating function on all classes. This is typically the case when several users (classes) can access the same set of objects (items) with varying quality of service (profits) but the cost of accessing them differs (weights). We explicitly denote the profits generated via this way as:

$$p_{ij} = \mathcal{R}\left(\mathcal{U}\left(p_1^{\min}, p_1^{\max}\right)\right). \tag{15}$$

Here, $\mathcal{R}$ signifies Reproducing the first generated profit vector for other classes. The second way to apply the generating functions is to take into account the class index i when deciding the interval from which the values are taken for each class, *e.g.* $\mathcal{U}(10(i-1), 10i)$ or $\mathcal{L}(10(i-1), 10i)$. When the uniform generating function is applied this way, the resulting profits in each class is still randomly selected but profits of different classes are dispersed into different intervals. While the linear generating function is applied, the profits are linearly assigned in different intervals. We denote this special application of generating functions as:

$$p_{ij} = \mathcal{C}(F), \tag{16}$$

where F is a generating function with different parameters for different classes and $\mathcal{C}$ signifies that the generating function is Class-dependent.

4.2. Generating the Weights

To generate the weights, we can apply a certain correlation on the generating function for each dimension. In particular, we define the following generating functions.

Uncorrelated Generating Function. In uncorrelated generating function, we simply assign weights with values uniformly and randomly selected within a certain

interval:

$$w_{ijk} = \mathcal{U}\left(w_{ik}^{\min}, w_{ik}^{\max}\right). \tag{17}$$

Weakly Correlated Generating Function. This generating function is motivated by previous results on KP instances [18]. The motivation is to slightly associate profits to weights, but still with a degree of freedom for each dimension. In our proposal, weights are assigned according to:

$$w_{ijk} = \mathcal{U}\left(\max\left(0, p_{ij} - \frac{p_i^{\max}}{\delta}\right), p_{ij} + \frac{p_i^{\max}}{\delta}\right). \tag{18}$$

We will use the following shorthand notation:

$$w_{ijk} = \mathcal{W}(\delta). \tag{19}$$

Strongly Correlated Generating Function. Strongly correlated generating function is also motivated by previous results where the correlation between profits and weights is strong:

$$w_{ijk} = p_{ij} + \frac{p_i^{\max}}{\delta}. \tag{20}$$

We use the following short hand notation for this function:

$$w_{ijk} = \mathcal{S}(\delta). \tag{21}$$

Inversed Strongly Correlated Generating Function. For inversed strongly correlated generating function, weights are assigned according to:

$$w_{ijk} = p_i^{\max} - \frac{p_{ij}}{\delta}, \tag{22}$$

and will be referred to as:

$$w_{ijk} = \mathcal{I}(\delta). \tag{23}$$

Note that the inversed strongly correlated generating function is not interesting to be used alone. Interesting cases occur when both strongly correlation and inversed strongly correlation coexist on different weight dimensions. Intuitively, these instances are hard to solve because careful trade-off between weights across dimensions has to be made. Although we are not aware of any realistic MMKP problems of this type, they are still interesting from a theoretical point of view.

Note also that the weights should always be non-negative, so we propose to build strongly and inversed strongly generating functions with different patterns. One drawback in these two definitions is that they do not represent a perfect symmetric case, *i.e.* the strongly and inversed strongly generating functions may not generate increasing and descending weights, respectively, with the same step length when δ is set to the same value. Other definitions are possible and it would be interesting to explore their properties, however, we leave them for future study.

Applying the Weight Generating Functions. Similar to the profit generating functions, one could apply the same generating function with the same parameter to all dimensions. But it is also possible to apply the same function with different parameters or even different generating functions for dimensions. In addition to simply applying the same generating function with the same parameters on all dimensions, here we propose two ways to apply the weight generating functions. The first one is to include the dimension index k as a parameter of the generating function so that the weight for a dimension k can be chosen in a range that depends on k for the uniform generating function, or the parameter δ can be a function of k for weakly, strongly and inversed strongly correlated generating functions. It is convenient to use a shorthand as follows:

$$w_{ijk} = \mathcal{D}(F), \tag{24}$$

where F can be, for example, $\mathcal{U}(1, 10k)$ for the uniform generating function, or $\mathcal{W}(k+5)$ and $\mathcal{S}(k+5)$ for weakly correlated and strongly correlated generating functions, respectively. Here, $\mathcal{D}$ signifies that the generating functions are Dimension-dependent. We could also apply different generating functions to different dimensions, for example, we will generate instances with the inversed strongly correlated generating functions on some dimensions and strongly correlated generating function on others. Under this case, we denote:

$$w_{ijk} = \mathcal{D}(F_1 F_2 \ldots), \tag{25}$$

where $F_1, F_2, \ldots$ are the generating functions in used.

4.3. Generating the Knapsack Capacities

Finally, knapsack capacities are generated by extending (5) to multiple classes and multiple dimensions. We generate a series of the S instances and the capacity of the kth dimension of the hth generated instance ($h = 1, 2, \ldots, S$), denoted as c_k^h, is dispersed from the minimum possible weight to the maximum possible weight:

$$c_k^h = \frac{h}{S+1}\left(\sum_{i=1}^{m} w_{ik}^{\max} - \sum_{i=1}^{m} w_{ik}^{\min}\right) + \sum_{i=1}^{m} w_{ik}^{\min}. \tag{26}$$

The parameter h will also be referred to as the *capacity level* of the instance in the series. Note that all the S instances do not need to have the same items (profit and weight assignment). However, in order to investigate the impact of the capacity level on the solution time in this paper, we let all S instances in the same series have the same profit and weight values. As a result, instances in a series differ from each other only by their capacities.

4.4. Summary of Instance Notations

We denote G-x-y a group of instances. The parameter x indicates the generating function that is chosen to allocate the profits, so in this paper x should be picked in $\{\mathcal{U}, \mathcal{L}, \mathcal{R}, \mathcal{C}(\mathcal{U}), \mathcal{C}(\mathcal{L})\}$. In the same idea, the parameter y indicates how the weights are computed. The set of generating functions considered in this paper is $\{\mathcal{U}, \mathcal{W}, \mathcal{S}, \mathcal{D}(\mathcal{U}), \mathcal{D}(\mathcal{W}), \mathcal{D}(\mathcal{S}), \mathcal{D}(\mathcal{SU}), \mathcal{D}(\mathcal{SI}), \mathcal{D}(\mathcal{SUI})\}$. And for the group names, we use the corresponding normal fonts instead of the calligraphic fonts used for the generating functions. For example G-U-W stands for the group of instances with profits generated with the uniform generating function $\mathcal{U}$ and weights generated with the weakly generating function $\mathcal{W}$. Among all combinations, we focus on the instances that either exhibit an interesting behavior, or appear to be standard families of instances. This subset of instance families are detailed in Table 2. Note in particular that we create two groups of instances using the $\mathcal{I}$ generating function for some dimensions. These two combinations have been chosen because, as we will show later, they exhibit

especially interesting hardness nature. Other combinations are obviously possible and hard instances other than those discussed in this study must exist. Discovering such instances could be an interesting future work.

5. Experiment Study

5.1. Experiment Setup

We implemented the BBLP algorithm with C++ programming language and built the binary with GNU g++ version 4.3.0. For the standard solvers, we employed the ILOG CPLEX version 11.2.0 [1] and GNU GLPK version 4.31. For both solvers, we keep the default parameters related with the algorithms.

All experiments have been carried out on the same computation platform, which is a Fedora 7 running on an IBM Thinkpad with an Intel Pentium M processor at 1.86GHz and with 1GB memory and 1GB swap space on the hard disk.

We generate the instances described in Table 2 with the proposed method, then we challenge the algorithms with these instances. In particular, we generate instances for $P(10, 5, 5)$, which are of the same size as I2 from the OR benchmark library. We have similar results for $P(5, 5, 5)$ and $P(15, 10, 10)$ which correspond to I1 and I3, respectively. However, the former is so easy that the differences are too small, while for the latter, most instances we generated can not be solved in reasonable time. Therefor, only results for $P(10, 5, 5)$ are presented in the paper. Since both the number of input variables and the values that the variables take have impact on the solution time of an instance, we select the variable values within the same range as I2 so the effects of different variable values are minimized. Notice also that some of the generating functions previously defined have random factors. For groups using these generating functions, we generate 20 series for each group to account for the random effects. For the groups that contain

[1] The CPLEX is licensed to "AMPL Student Edition", which is able to solve problems with up to 300 variables and this is enough for our example problems.

only the deterministic generating functions, *e.g.* linear profits with strongly correlated weights, the parameters of the generating functions determine the uniqueness of the instance, so only one series is evaluated. For each series, we generate 100 instances, *i.e.* $S = 100$. The instance generating program and the instances are available at `http://enstb.org/~gsimon/Resources/MMKP/`.

Some generated instances may be infeasible while others may be too hard to be solved to optimal in reasonable time. For the latter case, the execution time of the algorithm is limited to 600 seconds. As a result, the solution time that will be presented in the following part of this section could be the time for either obtaining the optimal solution, or asserting infeasibility, or the time used when the algorithm is aborted.

5.2. Average Solution Time

We first give an overview of the solution time of the generated instances to highlight the existence of hard instances. In Table 3, both average and maximum solution time is presented where the average is taken across capacity levels and across multiple series, and the maximum is taken from the average values across multiple series. Comparing with the solution time of I2 presented in Table 1, we find that certain groups of instances such as G-L-W, G-L-S, G-L-D(W) and G-L-D(S), *etc.* are much harder.

We can roughly classify these instances into three categories as indicated in the three separated parts in Table 3. From top to bottom, results for instances generated with uncorrelated, weakly correlated and strongly correlated generating functions are listed and a clear trend of increasing solution time could be observed. We conclude that high correlation between weights and profits generally makes an instance harder. If the profits are chosen according to the linear generating function, instances with weakly and strongly correlated weights become very hard even for the advanced solvers such as CPLEX and GLPK.

5.3. Capacity Level and Solution Time

Now we show the relationship between capacity level and solution time of the instances.

Figure 1 presents the solution time of G-U-$*$ instances according to the capacity level. We can observe that for uncorrelated cases, instances with lower capacity levels are generally very easy while hardest instances emerge at capacity levels between 40 and 50. The easiest uncorrelated instances with lower capacity level are due to their infeasibility while those with highest capacity level are trivial. While on the other hand, for weakly and strongly correlated cases shown in the middle and right most plots in Figure 1, respectively, the hardest instances usually appear at the center of the capacity level. Similar observations could be made from the G-L-$*$ cases in Figure 2. However, when the linear generating function is used, the weakly and strongly correlated instances become harder.

Figure 2 demonstrates also high variability of relative hardness within one series. This is especially obvious for the G-L-W/S instances. Some non-trivial strongly correlated instance could be extremely easy for CPLEX, GLPK and sometimes also for BBLP. The very special combination of capacity level, profit and weight admits very efficient branch-and-bound operation. Furthermore, thanks to the special mechanisms employed by GLPK and CPLEX, these instances can be solved even faster. These mechanisms consist of pre-process that may possibly reduce the number of variables, various branching heuristics and various cutting algorithms. By applying the default parameters of CPLEX and GLPK, these advanced algorithms are enabled and both solvers apply them dynamically during the search process. However, it is quite surprising that BBLP is generally faster than GLPK on strongly correlated instances, and it even achieves similar performance as CPLEX does on weakly correlated instances. Both imply that the additional efforts taken by CPLEX or GLPK do not help much in solving these instances.

The positions of hard instances are hard to predict when certain correlations exist. Notably, the G-U-W/S, G-L-W/S and G-R-W/S (shown in Figure 3) generally have similar properties that the hardest instances appear at 50% capacity and the advanced algorithms could solve certain instances very quickly. However experiments on G-R-D($*$) and G-L-D($*$) (in Figure 4 and Figure 5, respectively) show different properties. For example, the inversed strongly correlated dimen-

sion gives a clear cut on the feasible instances, making the hardest ones appear at a shifted position. The hardest instances of G-R-D(SU) and G-L-D(SU) appear also at positions shifted to the higher capacity levels, as shown in the left most figures of Figure 4 and Figure 5. Therefore, a rule of thumb is to use the whole series to benchmark the algorithms, instead of with only a few samples.

5.4. Non-trivial Infeasible Instances

The instances may be infeasible and they appear often in uncorrelated cases. As we show in Figure 7, if an instance is infeasible, it is generally easy for all the three algorithms to detect this fact, partially due to the fact that the LP relaxation for these instances are also infeasible. However, there exist infeasible instances that are *non-trivial* to detect. The same observation has also been claimed in [5]. These hard infeasible instances usually appear at intermediate capacity levels at which both infeasible and feasible instances exist.

5.5. The Critical Dimension

In Figure 8(a), we could see that the solution time increases linearly with the number of dimensions for the considered $P(10, 5, *)$ G-L-U instances. However, if the dimensions have mixed correlation properties, one generating function can be dominant, and reduce the impact of the number of dimensions on the hardness of the instance. Actually, instances generated with the strongly correlated generating function are the hardest to solve. Figure 6 presents the solution time of instances with a single weight dimension and the results are very similar to that of the multidimensional cases shown in Figure 2. Especially for the strongly correlated cases, the similarity implies that multiple strongly correlated weight dimensions may not be necessary for a hard instance. Now we create instances where one dimension is strongly correlated and other dimensions are uncorrelated. We observe that the impact of the hardness of the strongly correlated dimension diminishes when the number of uncorrelated dimensions increases. The explanation is that the number of items that are decided by the uncorrelated dimensions increases with the number of these uncorrelated dimensions,

so these items do not require to be decided by the strongly correlated dimension. Actually, it also means that the number of dimensions required to make the instance easy depends also on the capacity of these additional dimensions. The more constraining are the capacities of the additional uncorrelated dimensions, the fewer dimensions are probably required to make the instances easy. As extreme cases, adding unbounded dimensions on which the knapsack has unlimited capacity does not help at all while adding infeasible dimensions on which the knapsack has very limited capacity renders the instance infeasible regardless of the contributions from other dimensions.

6. Conclusion

We have proposed systematic methods to generate more comprehensive MMKP instances for benchmarking the algorithms. Several categories of MMKP instances have been produced to demonstrate that some MMKP instances are hard. Experiments on these hard instances with present exact algorithm and solvers also revealed some special structure of the problem. Briefly, the instance is hard to solve when all classes contain the same profit vector and the weights are correlated with the profits. Certain categories of instances are very hard for all considered algorithm and solvers: BBLP, GLPK and CPLEX, even though many advanced branching and cutting algorithms are employed by the two generic solvers. These instances contain classes with exactly the same set of items, making the instance symmetrical. It is known that symmetry makes integer linear programming hard. While many studies exist trying to break the symmetry, for example [23, 24, 25], they have not been applied to the MMKP. It could be an interesting direction for our future works.

Acknowledgement

We thank the anonymous reviewers for their constructive comments that helped us a lot in substantially improving this study.

References

[1] C. Lee, J. Lehoczky, R. R. Rajkumar, D. Siewiorek, On quality of service optimization with discrete QoS options, in: RTAS'99: Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium, IEEE Computer Society, Washington, DC, USA, 1999, p. 276.

[2] M. W. H. Sadid, M. R. Islam, S. M. K. Hasan, A new strategy for solving multiple-choice multiple-dimension knapsack problem in pram model, in: Asian Applied Computing Conference, 2005.

[3] S. Khan, Quality adaptation in a multisession multimedia system: Model, algorithms and architecture, Ph.D. thesis, University of Victoria (1998).

[4] S. Khan, K. F. Li, E. G. Manning, M. M. Akbar, Solving the knapsack problem for adaptive multimedia systems., Stud. Inform. Univ. 2 (1) (2002) 157–178.

[5] H. Kellerer, U. Pferschy, D. Pisinger, Knapsack Problems, Springer, 2004.

[6] R. Parra-Hernandez, N. Dimopoulos, A new heuristic for solving the multi-choice multidimensional knapsack problem, Systems, Man and Cybernetics, Part A, IEEE Transactions on 35 (5) (2005) 708–717.

[7] M. Moser, D. P. Jokanović, N. Shiratori, An algorithm for the multidimensional multiple-choice knapsack problem, IEICE transactions on fundamentals of electronics, communications and computer sciences 80 (3) (1997) 582–589.

[8] M. Hifi, M. Michrafy, A. Sbihi, A reactive local search-based algorithm for the multiple-choice multi-dimensional knapsack problem, Computational Optimization and Applications 33 (2-3) (2006) 271–285.

[9] M. M. Akbar, M. S. Rahman, M. Kaykobad, E. G. Manning, G. C. Shoja, Solving the multidimensional multiple-choice knapsack problem by constructing convex hulls, Comput. Oper. Res. 33 (5) (2006) 1259–1273.

[10] C. A. D. K. Harvey M. Salkin, The knapsack problem: A survey, Naval Research Logistics Quarterly 22 (1) (1975) 127–144.

[11] D. Pisinger, Core problems in knapsack algorithms, Oper. Res. 47 (4) (1999) 570–575.

[12] M. E. Dyer, J. Walker, Dominance in multi-dimensional multiple-choice knapsack problems, Asia-Pacific Journal of Operational Research 15 (2) (1998) 159–168.

[13] `http://www.ilog.com/products/cplex/`.

[14] `http://www.gnu.org/software/glpk/`.

[15] M. Hifi, M. Michrafy, A. Sbihi, Heuristic algorithms for the multiple-choice multi-dimensional knapsack problem, J Operat Res Soc 55 (12) (2004) 1323–1332.

[16] N. Cherfi, M. Hifi, A column generation method for the multiple-choice multi-dimensional knapsack problem, Comput. Optim. App. online first, Springer Netherlands (2008).

[17] A. Sbihi, A best first search exact algorithm for the multiple-choice multidimensional knapsack problem, Journal of Combinatorial Optimization 13 (4) (2007) 337–351.

[18] D. Pisinger, Where are the hard knapsack problems?, Comput. Oper. Res. 32 (9) (2005) 2271–2284.

[19] M. Chantzara, M. E. Anagnostou, Mvrc heuristic for solving the multi-choice multi-constraint knapsack problem, in: International Conference on Computational Science (1), 2006, pp. 579–587.

[20] `ftp://cermsem.univ-paris1.fr/pub/CERMSEM/hifi/OR-Benchmark.html`.

[21] S. Alam, M. Hasan, M. Hossain, A. Sohail, Heuristic solution of mmkp in different distributed admission control and QoS adaptation architectures for video on demand service, Broadband Networks, 2005 2nd International Conference on (2005) 896–903 Vol. 2.

[22] B. Han, J. Leblet, G. Simon, Query range problem in wireless sensor networks, to appear in IEEE communications letters.

[23] V. Kaibel, M. Pfetsch, Packing and partitioning orbitopes, Math. Program. 114 (1) (2008) 1–36.

[24] V. Kaibel, M. Peinhardt, M. E. Pfetsch, Orbitopal fixing, in: Proc. of the 12th Integer Programming and Combinatorial Optimization conference (IPCO), 2007.

[25] F. Margot, Exploiting orbits in symmetric ILP, Mathematical Programming, Series B 98 (1-3) (2003) 3–21.

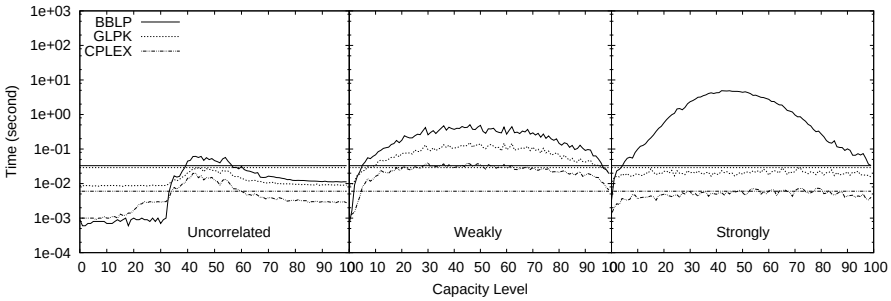


Figure 1: Solution time of G-U-$*$ instances.

Table 1: Solution time (second) of OR benchmark library instances I1 to I6.

Inst	m	n	l	CPLEX	GLPK	BBLP
I1	5	5	5	0.005	0.028	0.016
I2	10	5	5	0.006	0.029	0.033
I3	15	10	10	1.983	16.036	67.260
I4	20	10	10	31.045	1383.251	1532.059
I5	25	10	10	0.018	0.046	0.660
I6	30	10	10	0.204	0.190	2.369

Table 2: Generating Functions for Instances $P(10, 5, 5)$

Group	Profits (p_{ij})	Weights (w_{ijk})		
		Uncorr.	Weakly Corr.	Strongly Corr.
G-U-$*$	$\mathcal{U}(1, 50)$	$\mathcal{U}(1, 10)$	$\mathcal{W}(10)$	$\mathcal{S}(10)$
G-L-$*$	$\mathcal{L}(1, 50)$			
G-U-D($*$)	$\mathcal{U}(1, 50)$	$\mathcal{U}(1, 10k)$	$\mathcal{W}(k + 5)$	$\mathcal{S}(k + 5)$
G-L-D($*$)	$\mathcal{L}(1, 50)$			
G-C(U)-$*$	$\mathcal{U}(10(i - 1), 10i)$	$\mathcal{U}(1, 10)$	$\mathcal{W}(10)$	$\mathcal{S}(10)$
G-C(L)-$*$	$\mathcal{L}(10(i - 1), 10i)$			
G-C(U)-D($*$)	$\mathcal{U}(10(i - 1), 10i)$	$\mathcal{U}(1, 10k)$	$\mathcal{W}(k + 5)$	$\mathcal{S}(k + 5)$
G-C(L)-D($*$)	$\mathcal{L}(10(i - 1), 10i)$			
G-R-$*$	$\mathcal{R}(\mathcal{U}(1, 50))$	$\mathcal{U}(1, 10)$	$\mathcal{W}(10)$	$\mathcal{S}(10)$
G-R-D(SU)	$\mathcal{R}(\mathcal{U}(1, 50))$	$\mathcal{S}(10), \forall k = 1; \mathcal{U}(1, 10), \forall k \in \{2, 3, 4, 5\}$		
G-R-D(SI)	$\mathcal{R}(\mathcal{U}(1, 50))$	$\mathcal{S}(10), \forall k \in \{1, 2\}; \mathcal{I}(10), \forall k \in \{3, 4, 5\}$		
G-L-D(SU)	$\mathcal{L}(1, 50)$	$\mathcal{S}(10), \forall k = 1; \mathcal{U}(1, 10), \forall k \in \{2, 3, 4, 5\}$		
G-L-D(SI)	$\mathcal{L}(1, 50)$	$\mathcal{S}(10), \forall k \in \{1, 2\}; \mathcal{I}(10), \forall k \in \{3, 4, 5\}$		
G-L-D(SUI)	$\mathcal{L}(1, 50)$	$\mathcal{S}(10), \forall k \in \{1, 2\}; \mathcal{U}(1, 10), k = 3; \mathcal{I}(10), \forall k \in \{4, 5\}$		

Table 3: Solution Time (second) of Instances.

Group	CPLEX		GLPK		BBLP	
	Avg.	Max.	Avg.	Max.	Avg.	Max.
G-U-U	0.0051	0.0400	0.0124	0.0770	0.0168	0.2180
G-U-D(U)	0.0068	0.1190	0.0148	0.1820	0.0220	0.5179
G-C(U)-U	0.0047	0.0380	0.0130	0.0810	0.0180	0.3090
G-C(L)-U	0.0046	0.0430	0.0124	0.1470	0.0173	0.3210
G-C(U)-D(U)	0.0053	0.0580	0.0141	0.1140	0.0251	0.2690
G-C(L)-D(U)	0.0058	0.0700	0.0139	0.1040	0.0213	0.2620
G-R-U	0.0052	0.0410	0.0126	0.1130	0.0157	0.1930
G-L-U	0.0056	0.0500	0.0134	0.1020	0.0194	0.3350
G-L-D(U)	0.0070	0.0670	0.0143	0.1700	0.0241	0.6559
G-U-W	0.0243	0.2510	0.0789	0.5049	0.2277	1.6068
G-U-D(W)	0.0282	0.1940	0.0788	0.7339	0.2382	2.4456
G-C(U)-W	0.0105	0.0310	0.0269	0.1560	0.3069	4.6063
G-C(U)-D(W)	0.0101	0.0390	0.0242	0.1840	0.2420	2.6426
G-C(L)-W	0.0121	0.0460	0.0310	0.2150	0.3371	4.8203
G-C(L)-D(W)	0.0118	0.0420	0.0304	0.2270	0.3295	4.7473
G-R-W	0.0437	0.7389	0.1239	1.3708	0.2940	6.9169
G-L-W	0.6814	23.6984	20.5166	598.0341	3.8108	61.2037
G-L-D(W)	0.3460	10.9843	4.2153	587.2337	2.0304	35.4606
G-U-S	0.0051	0.0360	0.0203	0.1130	1.5901	63.4574
G-U-D(S)	0.0094	0.0710	0.0268	0.1470	0.9968	31.5992
G-C(U)-S	0.0025	0.0100	0.0112	0.0350	2.4810	49.7544
G-C(U)-D(S)	0.0042	0.0130	0.0130	0.0810	1.1158	84.2942
G-R-S	0.0252	0.2810	1.1301	51.5342	56.0733	184.0700
G-R-D(SU)	0.0086	0.1770	0.0418	0.8419	35.0805	161.3765
G-R-D(SI)	0.0036	0.0220	0.3080	29.4125	45.3696	169.8772
G-L-S	0.0963	0.6049	227.6951	600.0000	44.0825	186.1967
G-L-D(S)	0.0386	0.7069	300.3657	598.0401	50.9872	186.3017
G-L-D(SU)	7.1919	117.7880	113.2350	594.3146	13.5551	101.1256
G-L-D(SI)	0.0503	0.6669	80.9059	555.7105	14.1198	152.3548
G-L-D(SUI)	9.4867	235.9370	79.9511	592.1970	12.6077	137.0432
G-C(L)-S	35.7737	384.3760	154.1470	597.6111	68.4683	209.6621
G-C(L)-D(S)	25.6856	422.6000	158.5421	598.0471	60.3072	209.4772

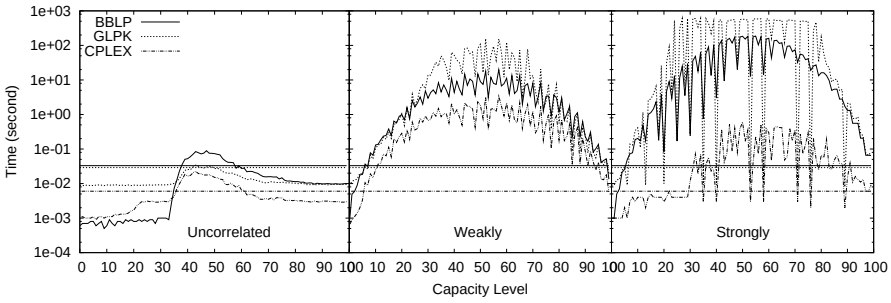


Figure 2: Solution time of G-L-$*$ instances

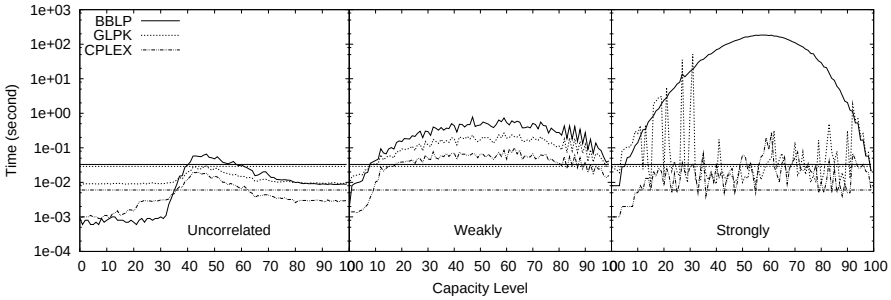


Figure 3: Solution time of G-R-$*$ instances.

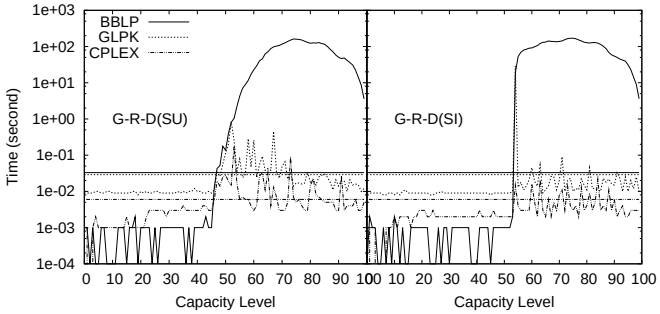


Figure 4: Solution time of G-R-D($*$) instances.

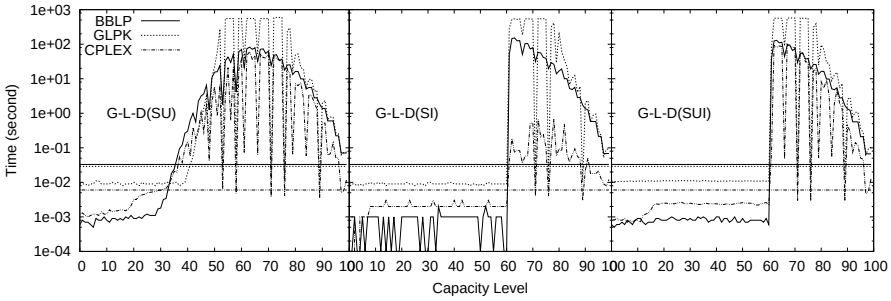


Figure 5: Solution time of G-L-D(∗) instances.

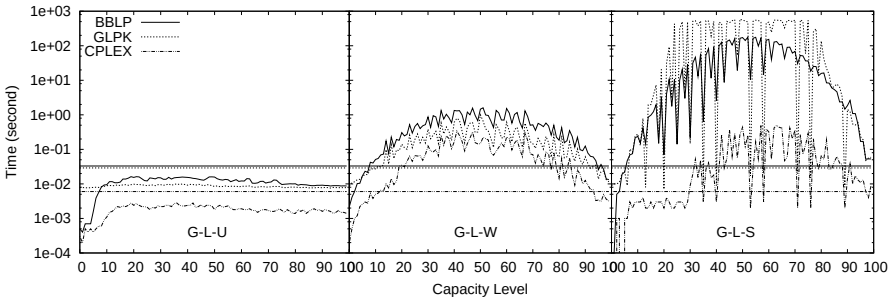


Figure 6: Solution time of single dimensional G-L-∗ instances.

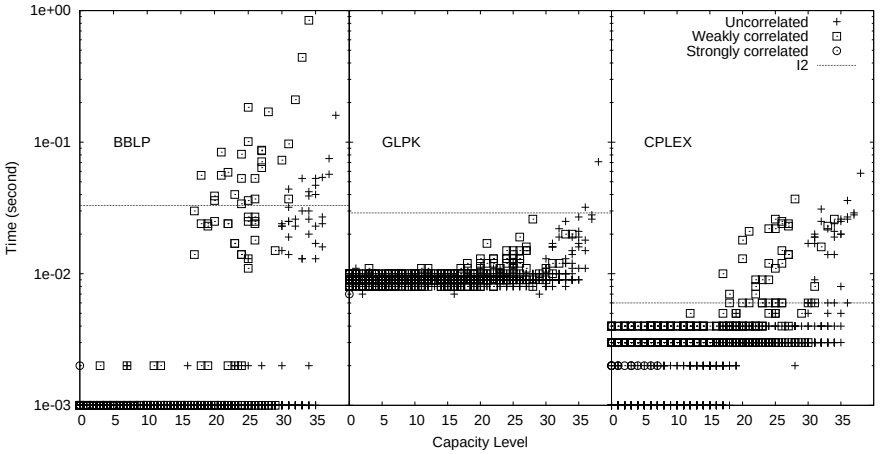
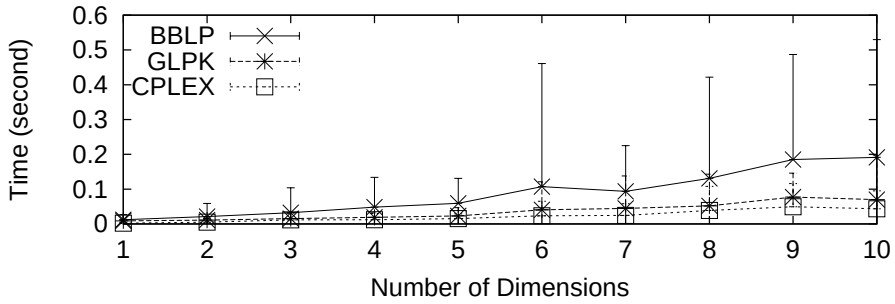
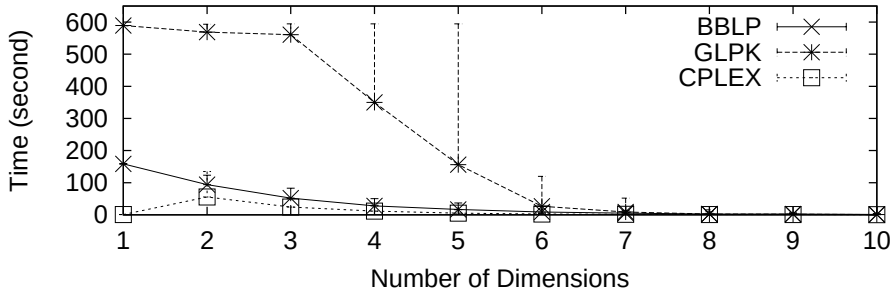


Figure 7: Nontrivial infeasible instances, G-C(U)-D(∗) as an example.



(a) G-L-U instances



(b) G-L-D(SU) instances, with a single strongly correlated dimension.

Figure 8: Solution Time vs. Number of Dimensions for $P(10, 5, *)$ instances.